Report for the Project work in languages and algorithms for artificial intelligence

Pozzoli Davide

GitHub:
https://github.com/treyvian/Log_Analysis_with_Spark

Artificial Intelligence
University of Bologna
Italy

Contents

1 Abstract

This PDF corresponds to the report for the 3 CFU Project work in languages and algorithms for artificial intelligence. This project aims to create a server log files analyzer written in Scala with the help of Spark, that given in input the text file containing the logs, the application will return some statistics on it in the form of csv files. The project can be run in local mode or through the google cloud platform and the results can be downloaded and graphed thanks to the 2 python scripts present.

2 Project structure

The main function is contained inside the *App.scala* file in *src/main/scala/*. The other two files presents are the inside the folder utils and are respectively:

- *AnalysisFunctions.scala* which contains the function for the analysis of the logs
- *Utils.scala* which contains the help functions for the main function. This file include for example the function for cleaning the input file.

The input file should be placed inside the folder *data* in the main directory
The folder *data_visualization* contains the python scripts for the download and visualization of the results after the analysis is performed.

More information about the execution of the project can be found in the *README.md* file

3 Project description

3.1 Input Data

As specified in the Abstract section the data in input is expected to be a server log files from a web server. The logs must be formatted following the Common Log Format. Each line in a file stored in the Common Log Format has the following syntax:

host ident authuser date request status bytes

The dash (-) in a field indicates missing data. for example in this case:

127.0.0.1 user-identifier frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326

- *127.0.0.1* is the **IP address** of the client (remote host) which made the request to the server.
- *user-identifier* is the **RFC 1413** identity of the client. Usually "-".
- *frank* is the **userid** of the person requesting the document. Usually "-".

- *[10/Oct/2000:13:55:36 -0700]* is the **date**, **time**, and **time zone** that the request was received, by default in strftime format
- *"GET /apache_pb.gif HTTP/1.0"* is the **request line** from the client. The method GET, /apache_pb.gif the resource requested, and HTTP/1.0 the HTTP protocol.
- *200* is the **HTTP status** code returned to the client. 2xx is a successful response, 3xx a redirection, 4xx a client error, and 5xx a server error.
- *2326* is the **size of the object returned** to the client, measured in bytes.

Online the only dataset that I was able to find which respects the common log format is this one. The dataset when unzipped weights around 3 GB. The idea was to find an even bigger dataset to have better results but it was the only one I was able to find.

In the logs all the *ident* and *authuser* were not present (-), so during the clean up the I assume that these two fields are always blank(-).

3.2 Cleanup data

Since the data in input is a text file once imported in a spark dataframe it will end up all in a single column where each row corresponds to an entry in the log file formatted as a single string. I used Regular Expressions to split each component of the log row in it's own column and in the mean time parse the timestamp in a format so it can be recovered more easily in the future.

Both these operation can be found in the file utils.py, which at the end returns the *cleanDF* which is the result of the format operations. The resulting dataframe has in the end 5 columns:

- **host**: String corresponding to the IP address of the client requesting the resource
- **timestamp**: a Spark Timestamp type containing the date and time of the log
- **status**: Integer corresponding to the HTTP code
- **path**: corresponding the path of the resource requested
- **content_size**: integer corresponding to the size of the object returned

3.3 Analysis functions

After the data has been cleaned the function for the extraction of useful information can be applied. I implemented a few but many more can be defined depending on the needs. They can be found in the file *AnalysisFunctions.scala*.

The ones that I decided to use are:

- *frequentHosts*: sort the number of hosts that did a request on the server ordered by the most frequent requester
- *frequentPath*: Finds the most frequent paths requested on the server ordering them by the number of times they are requested
- *httpStatusStats*: the status values that appear in the data and how many times
- *contentSizeStats*: some statistics about the sizes of content being returned by the web server. In particular, the average, minimum, and maximum content sizes
- *uniqueHostsCount*: Finds how many unique hosts are present in the logs

3.4 Saving dataframes

The dataframe containing the results of the analysis are saved inside a csv file in the output folder specified as an argument in the command line.

I am used to pySpark (the Python version) in which, when you want to write the dataframe to file you use the *pandas* library, which gives you the options to also specify the name of the file you are saving. This option with Scala version is not present, or at least I wasn't able to find it, and the closest solution was to rename the file after was saved, which is not ideal. Since the file is saved with the name automatically generated by spark in order to avoid problems in distinguish the files later in the output folder I create another folder called *analysisTIMESTAMP* where TIMESTAMP is the timestamp of the execution, in this way a unique folder is created each time the program is executed. Inside this folder I create one folder for each of the dataframe I'm saving with the name of the method from which the data comes from. Inside this last folder I save the final csv.

The method **coalesce(1)** put before calling write() when saving the results is necessary to have a single csv in output. In spark each partition is saved individually, so saving it without coalesce would result in multiple csv files. This remove parallelism since all the partitions are regrouped into a single worker, which is not a problem in this case since we have finished our analysis but it can be also dangerous, if there is too much data between workers grouping it all together would result in a crash.

3.5 Visualization

The initial idea was to write everything in Scala, but I incurred in 2 main problems:

Graphical library availability

I was looking for something simple to represents the data results something similar to matplotlib in Python. I started looking around and the two library

found compatible with what I had in mind were, Vegas and Plotly. In the end I wasn't able to make neither of them work. Vegas is not available for Scala 2.12 (the version I'm using) so I tried to downgrade the project to make it work but without success, Plotly should be compatible with the latest version of Scala but also this last one would not work.

The problem with both was that even though I was able to correctly import and compile them with Maven, the classes were not recognised correctly during the run time execution.

GCP API

The second problem was that Google Cloud Platform does not have an API in Scala for downloading files from the storage and the Buckets via code. The API is present in many languages including Python and Java that could theoretically work in a Scala file but I decided in the end to write this last part in Python to avoid creating more confusion.

00_download_data.py

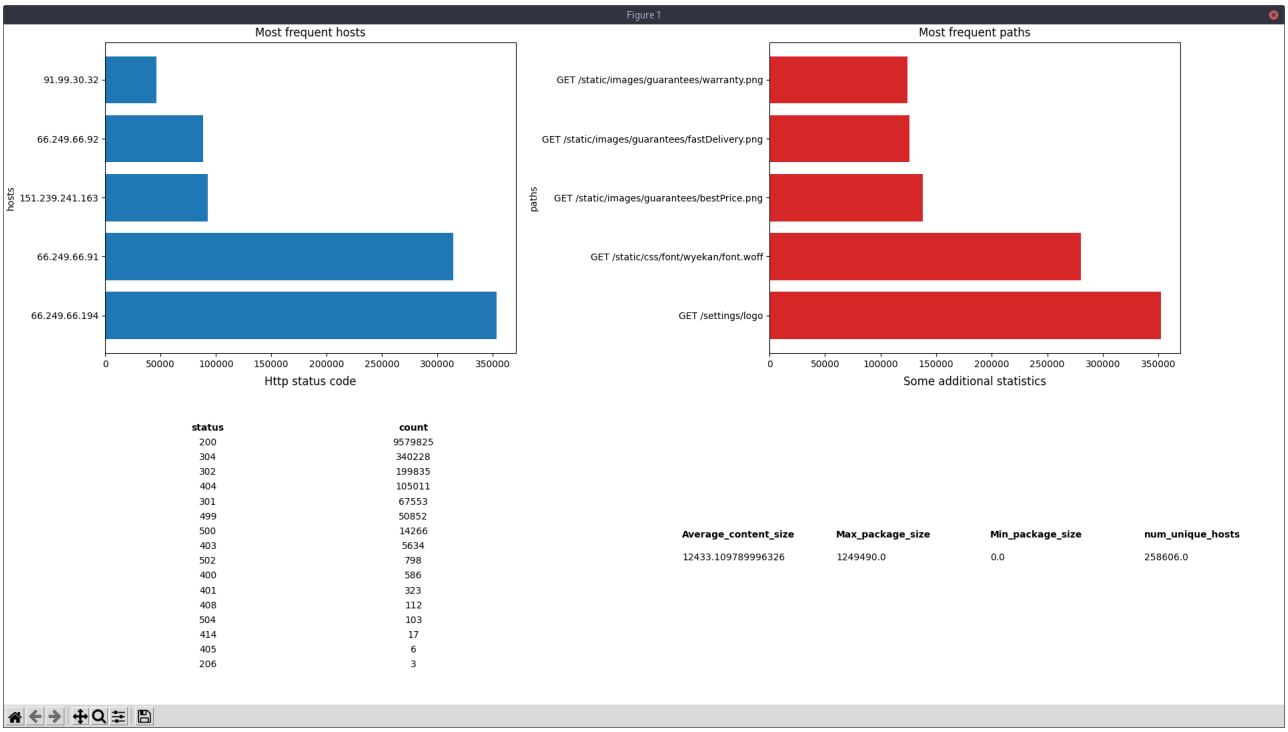
This file once executed will download data from the bucket in GCP and saves it in the folder */data_visualization/data/*. To execute it the json file for the service account authenticator must be present and the path specified in the file *constants.py* in the variable *KEY_JSON*.
I must be also executed with an argument

python 00_download_data.py –folder_name FOLDER_NAME

FOLDER_NAME is the name of the folder corresponding to the execution that created the data, which correspond to the folder inside $BUCKET_NAME/output. Es. *analysis22_07_04_22_30_34*.

01_plot.py

This file will plot the results downloaded with matplotlib in a very ugly dashboard.



This is just a proof of concept to not leave the results inside a csv but surely more works need to be carried out on the aesthetic part.

3.6 Results

For the results I recorded the time taken to execute the whole analysis, from right after Spark session is create to just before results are saved to file. In this way the time taken to create the session is not taken in consideration, especially because the session with *Yarn* takes approximately 1 minute to be set independently from the input.

The results are the average of 5 runs:

- *local*: 64s
- *GCP*: 46s

These results don't seem so great but I think this is due to two reasons. The first is that I wasn't able to create the cluster with the workers in the same region as the storage with the bucket, this of course create some delays when loading the data in Spark. Second the amount of data passed in input is too low, to really see a difference there is the need of a load of data that would create problem to the local executor. In this last case the usage of a cloud computing resources would make more sense. Unfortunately this file was the only one I was able to find.